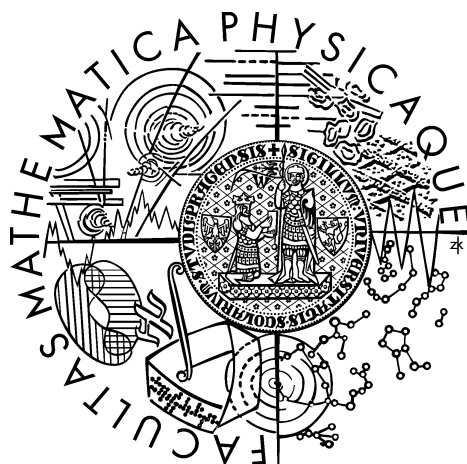


Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Adam Hraška

Distribúované kontejnery Distributed containers

Institution: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Study branch: Programming

2009

I would like to express sincere gratitude to my supervisor, RNDr. Filip Zavoral, Ph.D., for his immense patience throughout my work on this thesis. I would also like to thank my parents and my sister for their limitless support.

I hereby certify that I wrote the thesis myself, using only the referenced sources. I grant to Charles University permission to reproduce and to distribute publicly paper or electronic copies of this thesis and to grant others the right to do so.

Prague, August 10, 2009

Adam Hraška

Contents

1	Introduction	6
1.1	DHT operation overview	7
1.2	Thesis aim	8
1.3	Related work	9
1.4	Thesis organization	9
2	Key location	10
2.1	One-hop lookup	11
2.2	Kelips	13
2.3	Chord	16
2.4	Accordion	21
2.4.1	Budget management	22
2.4.2	Learning from lookups	23
2.4.3	Active exploration	24
2.4.4	Estimating entry freshness	25
2.4.5	Choosing eviction threshold	26
2.4.6	Improved lookup	27
2.4.7	Assessment	30
2.5	Summary	31
3	Data management	32
3.1	Data availability	32
3.1.1	Replica maintenance	34
3.1.2	Key database synchronization	35
3.2	Mutable data	39
3.2.1	Bamboo updates	39
4	Estimating latencies	41
4.1	The Vivaldi algorithm	42
4.1.1	Updating estimated error	43
4.1.2	Updating coordinates	44

5	Implemented DHT	46
5.1	User interface	47
5.2	Design	49
5.3	Implementation	50
5.3.1	RPCs and message transport	50
5.3.2	Lookup	50
5.3.3	Data maintenance	50
5.3.4	Kontajner	51
6	Summary	52
6.1	Conclusion	52
6.2	Future work	53
	Bibliography	55

Názov práce: Distribuované kontejnery

Autor: Adam Hraška

Katedra (ústav): Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. Filip Zavoral, Ph.D.

Abstrakt:

Distribuované hash tabuľky poskytujú funkcionality podobnú bežným hash tabuľkám. Na rozdiel od bežných hash tabuliek, DHT distribuujú uložené dáta v prostredí samoorganizujúcej sa peer-to-peer siete. V práci sa venujeme problémom spojeným s organizáciou DHT a algoritmom, ktoré tieto problémy prekonávajú. Zameriavame sa tak na algoritmy vyhľadávajúce v DHT, ako aj na algoritmy zabezpečujúce dostupnosť dát. Práca tak tiež obsahuje flexibilnú implementáciu jednoducho použiteľnej distribuovanej hash tabuľky. Tá je založená na algoritmoch poskytujúcich dobre vyvážený výkon v porovnaní s malým zaťažením sieťového pripojenia. Implementovaná tabuľka môže slúžiť ako základ rozsiahlej decentralizovanej aplikácie. Vyhodnotenie výkonu implementácie v skutočnom nasadení v takejto aplikácii je však mimo rozsah tejto práce.

Kľúčové slová: DHT, distributed storage, Accordion.

Title: Distributed containers

Author: Adam Hraška

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Abstract:

Distributed hash tables provide similar functionality to ordinary hash tables but they distribute stored data across a self-organized peer-to-peer network. In this thesis we explore the various challenges DHTs must face and we also examine the algorithms used to overcome them. We focus on key location algorithms as well as the data maintenance strategies. Furthermore, the thesis also includes a flexible and simple-to-use implementation of a DHT. The DHT is based on a set of algorithms which we believe provide good balance between performance and bandwidth usage. The implemented DHT may form the basis of a large-scale decentralized application. However, practical deployment of the DHT on hundreds of nodes and evaluation of the performance of the implemented DHT in such a deployment is outside the scope of this thesis.

Keywords: DHT, distributed storage, Accordion.

Chapter 1

Introduction

In recent years broadband internet connectivity is becoming increasingly widespread. With cheap high-speed internet connection available, users tend to spend more time using applications that involve many other people - such as massively multiplayer online games. As new personal computers are providing more computing power and memory every year, it is only natural to attempt to take the load off of the central servers of these massively collaborative applications and to distribute the load between the participants by harnessing the resources of client computers over the network.

This thesis aims to ease the development of such large-scale decentralized applications by implementing a simple-to-use *distributed hash table* (DHT). DHTs are capable of storing and retrieving $(key, value)$ pairs via a $PUT(key, value)$, $value = GET(key)$ interface. DHTs are designed to spread the load of storing and retrieving of these pairs across all the nodes participating in a DHT. $(key, value)$ pairs PUT into this data structure are automatically distributed between the participating nodes. DHTs organize their nodes in an overlay network in such a way that they are capable of locating and retrieving *values* with high probability in spite of nodes joining and leaving the DHT.

Since the first DHTs [24, 27, 29, 33] were introduced in 2001, many new ways of organizing nodes in a DHT have emerged. Although the various DHTs have different properties, they all provide a similar basic set of properties with a varying degree of success:

Decentralization There are no predetermined nodes that would store the entire $(key, value)$ database or that would even coordinate object location. Nodes self-organize in an overlay network and act as peers with no single node being overly responsible for the correct operation of a DHT.

Quick object location The overlay network is designed to enable prompt location of the node that stores a *value* associated with a given lookup *key*.

Operation under churn Despite nodes joining and leaving a DHT continually, it remains operational without significantly deteriorating the performance unless under heavy churn.

Fault-tolerance A DHT continues to function in the face of node failures.

Moderate network traffic Even though nodes are expected to be connected through a broadband connection, operation of a DHT does not saturate the network link, but leaves enough of bandwidth available for other application network activity.

Scalability DHTs retain the above properties even if there are thousands of nodes taking part in the DHT.

A data structure with the above properties could be used to store the large virtual worlds used in massively multiplayer online role-playing games. Although a player interacts with only a very limited portion of the virtual world, the world may potentially be gigantic. Even an instant messaging application can make use of a DHT by storing chat history and user profiles in it.

1.1 DHT operation overview

Conceptually, a DHT is composed of a data replication and maintenance layer on top of a key location layer.

The key lookup layer is responsible for assigning ownership of keys to individual nodes and for finding the owners of keys. This layer does not store any values associated with keys; it is only capable of returning the IP address of a node owning the searched-for key. To provide better support for the data replication layer, each node also maintains a small set of neighboring nodes. Upper layers are notified whenever this set of nodes changes due to nodes joining or departing the network.

Internally, the look-up layer maintains an overlay network of nodes. Each node is directly connected to a fraction of other nodes, but it is able to find the owner of any key in only a few hops. The way the nodes are interconnected in the overlay sets the upper limit on how quickly data can be retrieved and on how scalable and fault-tolerant the DHT is as a whole.

The data replication and maintenance layer is responsible for storing and fetching $(key, value)$ pairs from the nodes owning the *keys*. To ensure data availability in dynamic networks, $(key, value)$ pairs are replicated to some of the neighboring nodes. It is this layer's task to track the number of available replicas in the network and to add new replicas if some nodes fail. Furthermore, if we allow mutable data, newer versions of values must eventually eradicate old replicas.

If the amount of data stored is significant, replication can easily become the main contributor to network traffic overhead incurred by DHT operation.

1.2 Thesis aim

The goal of this thesis is to give an overview of the current ways of organizing distributed hash tables and dealing with the various issues that are connected with this decentralized data structure.

Secondly, this thesis aims to create a simple-to-use implementation of a distributed hash table in C++ in order to simplify the implementation of decentralized and possibly large-scale applications. The implemented DHT should be organized in such a way to meet the following requirements:

- Location of data should be as fast as possible in terms of the number of hops taken as well as the overall time spent searching.
- The DHT should potentially be able to scale up to thousands of nodes.
- It should be resilient to node failure. Lookup of *values* should not be affected by the failure of a small fraction of nodes.
- The overhead associated with the operation of the DHT should be minimal.

Some of these requirements are in opposition to each other, for example:

- Decreasing overhead by reducing the number of links to other nodes inadvertently increases the number of hops during lookups and it complicates recovery from node failures.
- Improving fault-tolerance and lookup times by introducing multiple redundant links to other nodes limits network's scalability because it adds overhead connected with updating these links as nodes join and leave the network.

In order to meet these requirements the author should familiarize himself with current approaches to structuring the various parts that compose a DHT. A combination of algorithms should be selected to best fulfill the goals set.

The thesis expects nodes to cooperate and need not consider malicious activity deliberately disrupting or otherwise subverting DHT's operation.

1.3 Related work

The first class of existing DHT implementations consists of prototype implementations by the authors of the individual algorithms. Except for Bamboo [26], these prototypes are not widely deployed or used. One may speculate it is because of the lack documentation or the fact that prototypes do not include a complete DHT system.

Another class are DHTs implemented for a single application [4]. However, they do not encapsulate the functionality in a library and are, therefore, harder to reuse.

Other notable DHT implementations include the Kademlia [21] protocol used in the BitTorrent file sharing network [2]. Nevertheless, the BitTorrent implementation uses the DHT only to locate other nodes and it does not provide a general way of storing values of various types or handling data replication.

1.4 Thesis organization

This thesis starts by examining various designs of the key lookup layer in chapter 2. The focus of chapter 3 is replica synchronization and the issues connected with mutable data. Next, we explore a way to predict network latencies in chapter 4. Chapter 5 describes the resulting C++ implementation of a complete DHT. Finally, we conclude this thesis with a summary in chapter 6.

Chapter 2

Key location

The sole purpose of the key location layer is to distribute ownership of keys between nodes and later to be able to locate the node owning a specific key. To do so in a scalable and efficient manner, key location algorithms employ the techniques of consistent hashing [17] to assign ownership of keys and exploit the small world phenomenon [18] to search for keys quickly.

Each node is assigned a random id from a large id space of, for example, 160-bit numbers. Keys also take the form of numbers chosen from this id space. Ownership of a key is then given to the node whose id is closest to the key in terms of a metric defined on the id space (e.g. a simple numerical distance). If both node ids and keys are uniformly distributed in the id space each node receives the ownership of approximately the same number of keys. What is more, whenever a node joins or leaves the hash table, the mapping of keys to nodes remains largely intact and only the mapping of a few keys with the smallest distance to the id of the node in question is affected. Therefore, unlike in traditional hash tables, increasing the size of the hash table does *not* result in rehashing of the entire table.

Applications typically require keys which are neither 160-bit numbers nor uniformly distributed. Nevertheless, application specific keys may be converted into keys of the id space using a cryptographic hash function such as SHA-1[15].

Routing tables are central to algorithms for finding the IP address of the owner of a key. A routing table contains the IP addresses and ids of nodes the local node is capable of directly communicating with. Every node stores in its routing table primarily the IP addresses of nodes nearby in the id space and only a few contacts to nodes that are very distant in the id space. Consequently, a routing table either contains the IP address of a search key's owner or it stores the IP address of a node which is closer to the search key and to which the search query can be forwarded. The closer a node is to a

key the more nodes in close vicinity of the key it knows about. Therefore, a search always progresses to nodes that are closer to the key in the id space until it reaches the node whose id is closest to the search key – the owner of the key. LOOKUP-SCHEME demonstrates this general searching scheme.

```

LOOKUP-SCHEME(key)
1  if OWNER-OF(key)
2      return this-node.IP-address
3  else
4      n = routing-table.GET-CLOSEST-NODE(key)
5      return n.LOOKUP-SCHEME(key)

```

A major problem is keeping the routing tables up-to-date in dynamic networks. When a node joins the DHT, it must fill its routing table and other nodes must eventually learn of the new node's arrival. Similarly, notification of node failures must propagate through the network to evict invalid routing table entries.

On the one hand, large routing tables offer better fault tolerance and reduce the number of hops during lookups. On the other hand, maintaining such tables leads to significant network traffic overhead because the more links to other nodes it contains the more links must be updated during DHT membership changes.

The rest of this chapter examines a number of key location algorithms. It is in no way meant to be an exhaustive list of the available lookup algorithms.

2.1 One-hop lookup

One-hop lookup [13] nodes maintain a complete routing table; i.e., the routing table contains an entry for every member of the DHT. As a result, nodes can reach the owner of a key in a single hop.

The identifier space consists of 128-bit numbers. Nodes are aligned along a circular identifier ring. Distance between two ids is then defined as the clockwise distance between the ids on the ring. For example, identifier 12 has a distance of 5 from id 7, but the distance of 12 from identifier 13 is $2^{128} - 1$ because to reach 12 from 13 in a clockwise direction one has to go around the entire ring. Therefore, a key belongs to a node if it falls between its id and the predecessor's id on the identifier ring.

```

ONE-HOP-LOOKUP(key)
1  return routing-table.GET-CLOSEST-NODE(key)

```

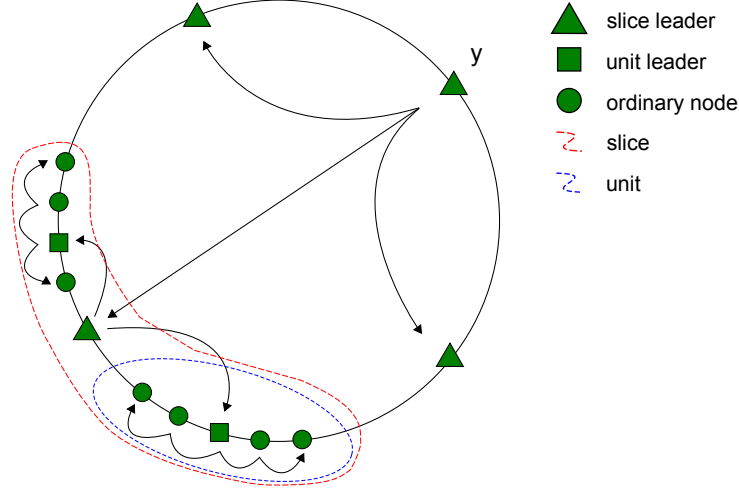


Figure 2.1: One-hop slice leader y sends out aggregate messages of membership changes in its slice to other slice leaders, which in turn propagate the message to their respective unit leaders. The notifications reach other nodes within a unit using the successor and predecessor links.

Looking up a key is as simple as searching the routing table. If the routing table entry returned is invalid due to node failure, the node may still attempt to locate the key at the key's next closest node, i.e., the successor of the failed node.

Updating routing tables

Because each node holds a reference to every other node in the DHT, routing tables of all nodes must be updated after DHT membership changes. Membership change notifications are disseminated using three level hierarchical trees. The circular identifier space is divided into a fixed number of slices. The successor node of the mid-point of the slice id space is appointed a *slice leader*. Similarly, slices are further subdivided into a fixed number of units and again, each unit has a dynamically chosen *unit leader* which is the successor of the mid-point of the unit id space.

Nodes regularly ping their successor and predecessor nodes on the id ring. If a node detects a new successor or if the current successor fails and stops responding, the node notifies the current slice leader of the event. The slice leader collects membership change notifications for a short period of time (t_{big}) before sending an aggregate message of all the local slice membership changes to other slice leaders. Slice leaders also wait to aggregate messages from each other before sending the notifications to the respective unit leaders

(t_{wait}). Notifications propagate within a unit by sending the information with regular ping messages.

The network traffic associated with updating all the routing tables is reduced by using dissemination trees. Furthermore, aggregating messages and piggy-backing information on the keep-alive messages reduces the overhead of sending small messages.

Assessment

Advantages of One-hop include:

- Swift lookup in only a single hop.
- Extremely fault-tolerant.

Disadvantages of One-hop may include:

- Significant load is placed on a couple of selected nodes — the slice leaders.
- Scalability is limited by the upstream bandwidth available to slice leaders.
- Memory requirements of storing the routing table of the entire DHT.
- It is necessary to estimate the size of the DHT and the expected churn rate beforehand and set the number of slices and units, and the length of t_{wait} and t_{big} accordingly.

One-hop appears to be a viable solution in smaller and stable environments where we expect a moderate DHT membership change rate.

2.2 Kelips

Kelips [14] is based on a loosely structured overlay network that provides lookup in a constant number of hops using $\Theta(\sqrt{n})$ sized routing tables.

Nodes are divided into a fixed number of affinity groups by means of a consistent hashing function. The number of groups, k , should be set to roughly \sqrt{n} to achieve optimal performance, where n is the number of participating nodes. Unlike in other DHT lookup algorithms, there is no further distinction between nodes of a group and any node within an affinity group may become the owner of any key that belongs to the group.

The routing table consists of addresses of all the nodes lying in the local affinity group. Furthermore, for each of the other groups it contains a small constant number of contacts to nodes in the other group. Apart from the routing table, every node maintains a complete catalog of keys stored in its affinity group. The catalog contains the addresses of current owners or *home nodes* of individual keys.

KELIPS-LOOKUP(*key*)

```

1  key-group = GET-AFFINITY-GROUP(key)
2  if key-group == local-affinity-group
3      return key-catalog.GET-HOME-NODE(key)
4  else
5      node = GET-RANDOM-GROUP-CONTACT(key-group)
6      return node.KELIPS-LOOKUP(key-group)
```

Kelips locates a key's home node by querying any node of the key's affinity group. The queried node searches its catalog to return the home node's IP address. Kelips copes with invalid routing table entries and non-responding nodes by querying a different node of the key's affinity group and if all contacts to the other group are stale it may still attempt to retry the query by executing it on another node in its local affinity group.

The Kelips algorithm requires keys to be inserted into the overlay network before they can be located. To insert a key, a node contacts any node in the key's affinity group which in turn forwards the insert request to a random node in its affinity group that becomes the key's home node. Forwarding insertion requests balances load within the affinity group. It is the home node's responsibility to inform other nodes in its group that it acquired the ownership of the key and that the key catalogs have to be updated appropriately.

Updating routing tables and key catalogs

Entries in the routing table and the key catalog have an associated heartbeat count. This counter needs to be updated regularly. Otherwise, the entry is considered to be no longer valid and it is evicted from the table. Therefore, nodes must periodically inform others of their existence and of the keys they own.

Kelips uses a gossip mechanism to refresh the heartbeat counts. Every couple of seconds a node selects a few neighbors in its affinity group and notifies them of its existence. These nodes forward the updated heartbeat count for a small number of rounds to other randomly selected nodes in the

group. Similarly but less frequently, nodes also notify other groups of their presence. According to [14] heartbeat updates reach all nodes of a group with high probability and with an expected latency of $O(\sqrt{n} \cdot \log^2(n))$ hops; updates reach nodes in other groups with an expected latency of $O(\sqrt{n} \cdot \log^3(n))$ hops.

In order to reduce network traffic, nodes devote only a constant amount of bandwidth to disseminate heartbeat updates. Consequently, if there are more updates than the bandwidth limit allows, new updates are given a priority over updates that have already been sent out in the previous rounds.

Locality awareness is built into the gossip protocol. Gossiping nodes measure their round-trip times and store the measurements in the routing table. Whenever a nodes needs to select a contact from the routing table, the random selection is biased toward nodes with lower measured round-trip times. As a result, key location as well as gossip dissemination is sped up.

Assessment

Advantages of the Kelips protocol include:

- Quick lookup in $O(1)$ hops. In moderately stable networks lookups take 2 hops.
- High fault-tolerance.

Disadvantages:

- Significant memory requirements – $\Theta(\sqrt{n})$ routing table size and $O(\sqrt{n})$ for the catalog of all the keys in the local affinity group.
- $O(\sqrt{n} \cdot \log^2(n))$ convergence time of key insertion and membership change.
- It is necessary to estimate the size of the DHT to divide nodes into the appropriate number of groups.
- Continuous background network traffic.

The Kelips protocol offers fast lookups, but under heavy network churn, convergence times might become an issue. What is more, the key catalog may grow noticeably large. Therefore, Kelips seems to be suitable for medium-sized networks where data repository size overshadows the size of the key catalog.

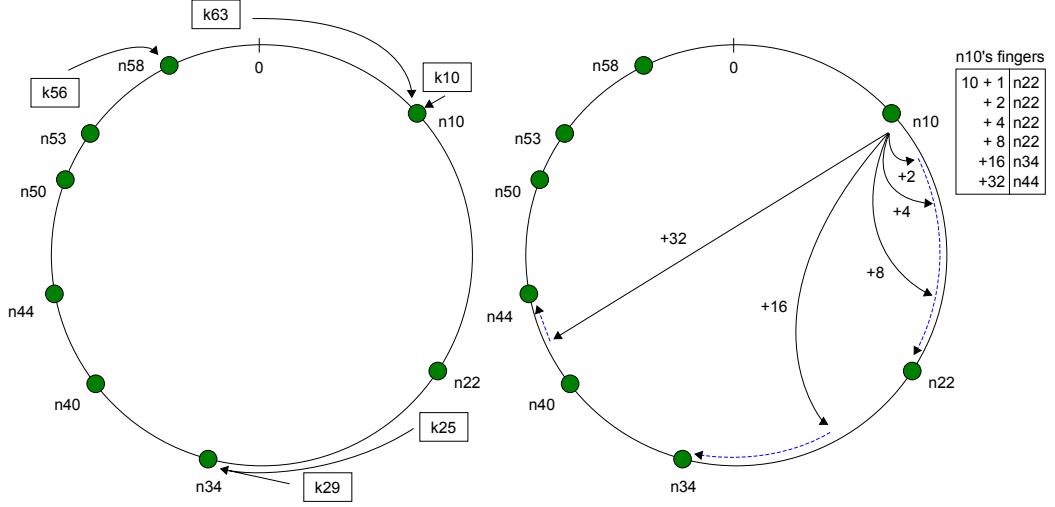


Figure 2.2: The left diagram depicts an identifier ring with $2^6 = 64$ identifiers. Node with id 34, $n34$, is responsible for keys 25 and 29. Notice that key 10 is owned by node 10 ($n10$). The right figure shows the routing table of node 10.

2.3 Chord

The Chord lookup algorithm [29, 30] offers a good trade-off between routing table size of $\Theta(\log(n))$ and search speed of $O(\log(n))$ hops.

Chord uses an identifier space organized similarly to One-hop's (section 2.1). m -bit identifiers (usually for $m = 160$) are placed on a circular ring and the distance between two identifiers is measured in the clockwise direction on the ring. A node receives ownership of a key if the key lies between the node's id and its predecessor node's id on the ring. In other words, a key's successor node on the ring is the owner of the key.

Routing tables closely follow the small world phenomenon [18] and store more contacts to nearby nodes than contacts to nodes farther away on the id ring. Routing tables comprise m entries. The i^{th} entry, sometimes called the i^{th} *finger*, stores a contact to the first node on the ring whose distance from the current node is at least 2^{i-1} . Therefore, the first routing table entry is a contact to the successor node ($i = 1, 2^0 = 1$) and the last entry stores a contact to a node that is placed at least half-way across the ring ($i = m, 2^{m-1}$ is half of the total number of 2^m identifiers). Generally, a node with id n maintains as the i^{th} routing table entry a contact to the successor node of id $n + 2^{i-1}$. Apart from the routing table entries, each node has a contact to the predecessor node.

CHORD-LOOKUP(*key*)

1 **return** FIND-SUCCESSOR(*key*)

FIND-SUCCESSOR(*id*)

1 **if** *id* == *this-node.id*

2 **return** *this-node*

3 **if** *id* \in (*this-node.id*, *successor.id*]

4 **return** *successor*

5 **else**

6 *node* = GET-CLOSEST-PREDECESSOR(*id*)

7 **return** *node*.FIND-SUCCESSOR(*id*)

GET-CLOSEST-PREDECESSOR(*id*)

1 **for** *i* = *m* **downto** 1

2 **if** *routing-table*[*i*] \in (*this-node.id*, *id*)

3 **return** *routing-table*[*i*]

4 **return** *this-node*

FIND-SUCCESSOR captures the lookup algorithm. A node forwards the search query to the node that most closely precedes the search key. The rationale is that nodes closer to the search key know more about the nodes in the key's vicinity. Search ends when it reaches the key's immediate predecessor.

Because the distances to fingers are powers of two, the search may always advance at least half-way across the remaining distance to the search key. As a result, if nodes are uniformly distributed along the identifier circle, searches complete in $O(\log(n))$ hops.

A notable property of the lookup algorithm is that correct successor contacts guarantee the correctness of the entire lookup procedure. Fingers are only used to speed up the process. Even if routing table entries are outdated and the i^{th} finger is no longer the *immediate* successor of $node-id + 2^{i-1}$, lookup progresses closer to search key's successor. Consequently, if the successor contact is maintained up-to-date, lookup cannot skip the key's successor and must eventually locate the owner.

Node joins

To ensure proper operation of the lookup procedure CHORD-LOOKUP even during node joins, nodes must keep a valid and possibly up-to-date successor contact. Nodes update their successor contacts by running STABILIZE periodically.

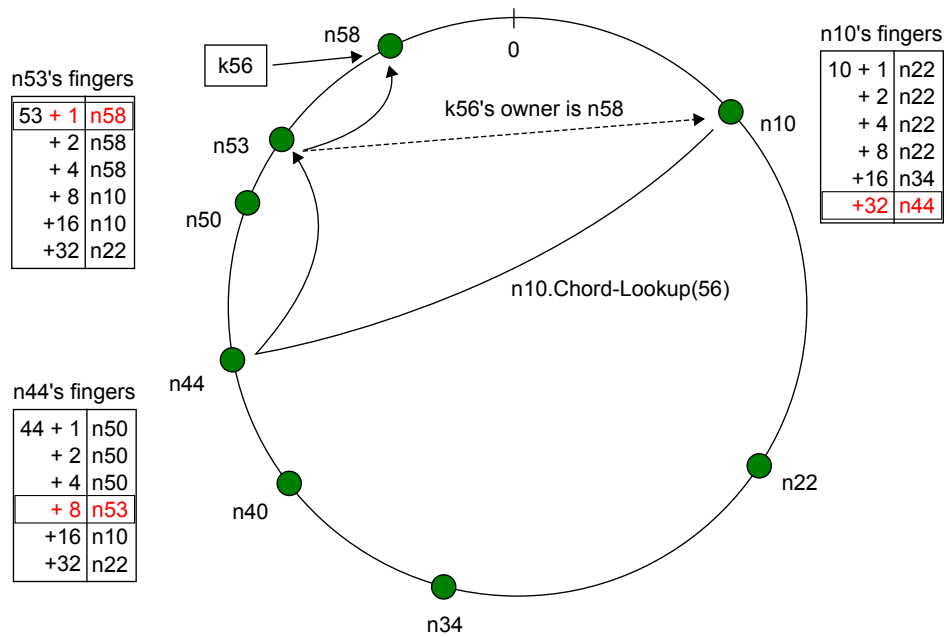


Figure 2.3: Example Chord lookup. Node 10 initiates the lookup of key 56. The closest preceding node of identifier 56 in the routing table is node 44. Node 10 sends the query to node 44, which in turn forwards the query to node 53, which is the closest preceding node of 56 it knows of. Node 53 happens to be the immediate predecessor of node 58, which most closely succeeds the search key 56, and is, therefore, the owner of key 56. The lookup succeeded and node 53 returns the address of the search key's owner, node 58, back to the query originator, node 10.

CREATE

```
1 predecessor = NIL
2 successor = this-node
```

JOIN(*bootstrap-node*)

```
1 predecessor = NIL
2 successor = bootstrap-node.FIND-SUCCESSOR(this-node.id)
```

STABILIZE

```
1 node = successor.predecessor
2 if node.id ∈ (this-node.id, successor.id)
3     successor = node
4 successor.NOTIFY(this-node)
```

NOTIFY(*node*)

```
1 if predecessor == NIL or node.id ∈ (predecessor.id, this-node.id)
2     predecessor = node
```

Successor contact stabilization after a node join is depicted in Figure 2.4. New node y initiates the join process by calling JOIN, which finds the correct successor, node z , and enables the new node to look up keys in the overlay. Next, node y notifies its successor z of y 's existence (STABILIZE invokes NOTIFY). Node z gets the chance to update the predecessor contact to the joining node y from the original predecessor, node x . Although y has a proper successor and z has already updated its predecessor contact, the joining node y is not yet reachable via the overlay. y becomes visible to the overlay once the original predecessor of z , node x , stabilizes its successor via STABILIZE. Once x updates its successor to y , the join process is complete.

Successor contacts remain valid during the entire join operation. Lookups in the overlay network are, thus, not disrupted in any way even in the face of concurrent joins.

Node leaves and failures

The Chord protocol relies on correct successor pointers. If the successor node fails it may be very inefficient or next to impossible to find the new successor using only the routing table and predecessor pointers.

In order to increase the robustness of the protocol, each node maintains a complete list of r successor nodes instead of a single successor. Therefore,

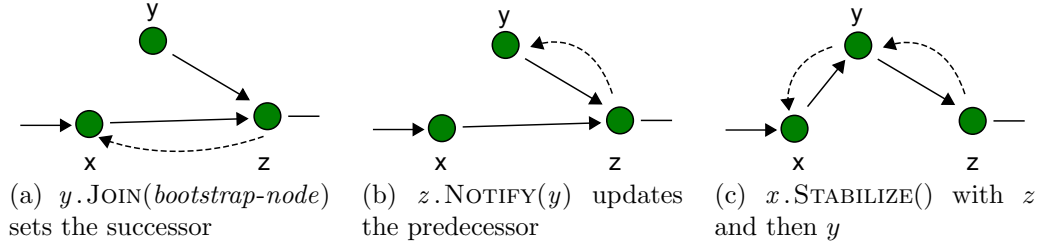


Figure 2.4: Chord join process. (a) The joining node y finds the correct successor contact z with the help of a not depicted bootstrapping node. (b) z .NOTIFY(y) updates the predecessor contact to y . (c) x .STABILIZE() stabilizes with x 's presumed immediate successor z and learns about the joining node y . y has successfully joined the overlay network. In turn, x stabilizes with y and y acquires the correct predecessor contact x .

if the immediate successor fails, the node can choose the next live successor in the list and continue normal operation.

To disrupt the overlay network, all r successors of a node would have to fail simultaneously. Assuming node failures are independent events, the probability of all r successors failing simultaneously decreases exponentially with r . Furthermore, neighboring nodes on the identifier ring are likely to be geographically far away due to their node ids being selected at random or generated by hashing the IP addresses.

The STABILIZE procedure is modified to accommodate changes to the protocol. It retrieves the successor's successor list. Then, STABILIZE removes the last entry and prepends the successor contact to form the local node's successor list.

Because a node actively monitors whether the nodes in the successor list are alive, it can rather quickly detect if these nodes leave the network or simply fail. Therefore, the successor list may serve the data maintenance layer as a set of nodes where to store replicated data.

Updating routing tables

Nodes periodically run FIX-FINGERS to update their routing tables. Routing tables need not be maintained as aggressively as the successor contacts; therefore, FIX-FINGERS is run less frequently than STABILIZE.

FIX-FINGERS

```

1   $next = next + 1$ 
2  if  $next > m$ 
3       $next = m$ 
4   $routing-table[next] = \text{FIND-SUCCESSOR}(this-node.id + 2^{next-1})$ 

```

Assessment

Advantages:

- Scalability.
- Reasonably small number of $O(\log(n))$ hops during lookups.
- Very small per-node soft state consisting of $O(\log(n))$ routing table entries and the successor list.
- Lookups remain largely unaffected in highly dynamic networks.
- Small overhead associated with membership changes.

Disadvantages:

- While lookup hop count remains small, the overall lookup time may noticeably increase especially if communicating with nodes that are physically very far away.

Chord offers a good compromise between lookup times, memory requirements and network overhead. What is more, the protocol is able to deal with heavy network churn efficiently.

2.4 Accordion

Accordion [20] builds on the successes of the Chord protocol (section 2.3). While it guarantees lookups in at most $O(\log(n))$ hops, it adapts to network dynamics and significantly reduces the number of hops needed to look keys up in stable non-changing networks.

Accordion's authors analyzed multiple designs of key location protocols and came to the following observations [19]:

- Parallelized lookups mask the large increase in lookup latency due to timeouts more effectively than checking the liveness of neighbors more often.

- Using bandwidth to increase routing table size is more effective at lowering overall lookup times than checking the freshness of routing table entries. Because routing tables may be reasonably fresh, pinging alive neighbors wastes bandwidth while searching for other neighbors reduces the overall hop count.
- Learning from lookup traffic (especially from message acknowledgments) is a more efficient way of acquiring new contacts than active exploration.
- The ability to control bandwidth consumption positively influences performance under different network sizes and scalability.

Accordion allows the user to set a single parameter, a network bandwidth budget. In order to meet the budget, it automatically adapts to varying network churn and size by adjusting the size of the routing tables all the while providing low latency lookups.

The size of the routing table is a result of an equilibrium between eviction and learning of node contacts. Instead of checking the liveness of neighbors via periodic keep-alive messages, Accordion estimates the liveness of nodes and evicts the nodes which it deems to have failed. On the other hand, new contacts are learned as a side effect of ordinary lookups.

Instead of imposing a rigid routing table structure like that of Chord, Accordion only specifies the desired density of neighbors with respect to distance and it does not require nodes from specific areas of the identifier space. Neighbors are selected so that the resulting density is inversely proportional to their distance from the node. Therefore, neighbors are selected with a $\frac{1}{d}$ distribution: a neighbor with distance d from the local node is chosen with probability proportional to $\frac{1}{d}$. This routing table structure allows flexible control over the size of the table, which can be adjusted along a continuum. Furthermore, the $\frac{1}{d}$ distribution is scalable and offers a $O(\frac{\log(n) \cdot \log(\log(n))}{\log(s)})$ lookup hop count where s is the size of the routing table (section 5.3 of [19]).

Accordion inherits the circular identifier space from Chord. Moreover, it maintains a successor list and utilizes the same node join protocol.

2.4.1 Budget management

Users specify the bandwidth budget with two numbers: the average desired network traffic in bytes per second, r_{avg} , and the maximum allowed burst size in bytes, b_{burst} .

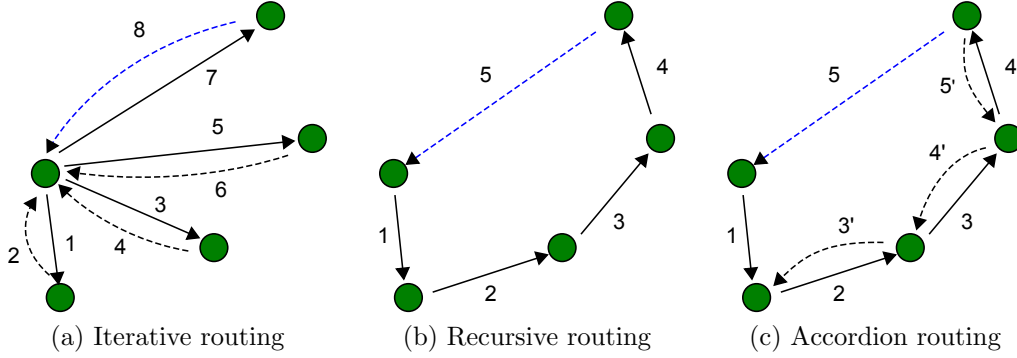


Figure 2.5: (a) In iterative routing the querying node controls the entire lookup process and contacts each node of the lookup path in order to obtain useful routing entries and find the next hop node. (b) Querying node only initiates a lookup. Each node of the lookup path determines the next hop node and forwards the query to it. In comparison with iterative routing, lookups are nearly twice as fast. (c) Accordion routes in a recursive fashion. Each forwarded query is acknowledged. Nodes learn from routing entries piggy-backed in the acknowledgments.

Each node keeps track of available bandwidth budget for active exploration in bytes, b_{avail} . Each time a node generates network traffic, it decreases b_{avail} by the size of the packets sent. Likewise, b_{avail} is decreased for any acknowledgments received. On the other hand, b_{avail} does not change due to unsolicited incoming traffic or due to acknowledgments sent out in response to such traffic.

b_{avail} is incremented every t_{inc} seconds by $r_{avg} \cdot t_{inc}$, where t_{inc} is the size of an exploration packet divided by r_{avg} (subsection 2.4.3). Nodes are allowed to decrease b_{avail} down to the minimum value of $-b_{burst}$. Whenever a node reaches the minimum value, i.e. $b_{avail} = -b_{burst}$, it stops sending any redundant or low priority traffic (such as exploration packets or parallel lookups). As a result, the average network traffic rarely exceeds r_{avg} .

2.4.2 Learning from lookups

Accordion uses recursive routing during lookups. Not only is recursive routing faster than iterative routing (see Figure 2.5), it also provides a way of learning new routing entries with the desired neighbor distribution.

While nearly all lookups end with a small hop to the key's immediate predecessor, only queries for keys far away start with long hops. As a result, most lookup hops are short. When node x forwards a search query for key k

to node y , node y sends back an acknowledgment. In order to make better use of network traffic and give the previous hop x a chance to learn new routing entries, node y also piggy-backs in the acknowledgment message a small constant number (e.g. 5) of routing entries that lie between y and the search key k in the identifier space. Because most lookup hops are relatively short, nodes learn primarily about nodes nearby.

GET-NODES-TO-LEARN($node, parallelism, key, m$)

```

1   $node-table = routing-table$ 
2   $node-table.EVICT(parallelism)$ 
3   $s = \text{nodes between } this-node.id \text{ and } key \text{ in } node-table$ 
4  // Find the  $m$  physically closest neighbors to  $node$  using estimates
5  // of network latencies between the neighbors and  $node$  (see chapter 4)
6  sort  $s$  by ESTIMATE-RTT( $node$ , entry in  $s$ ) in ascending order
7  return  $m$  first entries of  $s$ 
```

2.4.3 Active exploration

Learning new neighbors from lookups is efficient; however, search keys may not be uniformly distributed or nodes look keys up so infrequently that routing entries no longer follow the small world distribution. Therefore, nodes devote the budget left over from lookups to actively explore the identifier space and restore the appropriate neighbor distribution. Nodes send out exploration packets only if b_{avail} (subsection 2.4.1) is positive.

Assuming a $\frac{1}{d}$ distribution, the distance between neighbor x and the successive neighbor should be proportional to $d(this-node, x)$ which is the distance between the local node and x on the identifier ring. A node with id y calculates and compares the *scaled gaps* between its successive neighbors with ids n_i and n_{i+1} respectively:

$$g(n_i) = \frac{d(n_i, n_{i+1})}{d(y, n_i)}$$

The area of the identifier ring between two successive neighbors with the largest scaled gap $g(n_i)$ is in need of exploration because routing entries pointing to that area are the most sparse with respect to the small world distribution. Consequently, a node sends exploration packets to the node n_i with largest scaled gap as GET-NODE-TO-EXPLORE demonstrates. n_i , in turn, returns a small number of routing entries between n_i and n_{i+1} .

GET-NODE-TO-EXPLORE

```

1  // Only explore if there is budget left over from lookups
2  if  $b_{avail} \leq 0$ 
3      return NIL
4
5   $n_i = \text{routing-table.GET-SUCCESSOR}(\text{this-node.id})$ 
6   $n_{i+1} = \text{routing-table.GET-SUCCESSOR}(n_i.id)$ 
7   $s = \emptyset$ 
8
9  // Calculate the scaled gaps between successive neighbors  $n_i$  and  $n_{i+1}$ 
10 while  $n_{i+1} \neq \text{this-node.id}$ 
11      $n_i.gap = \frac{d(n_i, n_{i+1})}{d(\text{this-node}, n_i)}$ 
12      $s = s \cup \{n_i\}$ 
13     // Proceed to the next pair of neighbors
14      $n_i = n_{i+1}$ 
15      $n_{i+1} = \text{routing-table.GET-SUCCESSOR}(n_i.id)$ 
16 return node in  $s$  with largest  $gap$ 

```

2.4.4 Estimating entry freshness

The routing tables are periodically cleansed of contacts to nodes whose estimated probability of being alive is below a certain threshold $p_{threshold}$. The value of $p_{threshold}$ is set to minimize the number of hops during lookups.

To estimate the probability of a node being alive, node lifetimes are modeled with a Pareto distribution. Assuming node lifetimes follow the Pareto distribution, the probability of a node being alive for at most t seconds is:

$$\Pr(lifetime < t) = 1 - \left(\frac{\beta}{t}\right)^\alpha$$

where α and β are the shape and scale parameters of the distribution. Pareto distribution has been chosen because it approximates the node lifetime distributions measured in actual peer-to-peer networks such as the Gnutella network very well [28]. These measurements revealed that nodes which have been connected for a long time tend to stay online for an even longer period of time. Therefore, the lifetime estimate is not based only on Δt_{since} , the time which elapsed since the node was last known to be alive, but also on the time the node has already spent in the overlay network, Δt_{alive} . Thus, the probability of a neighbor being alive after Δt_{since} seconds without contact is:

$$\Pr(lifetime > (\Delta t_{alive} + \Delta t_{since}) \mid lifetime > \Delta t_{alive}) = \left(\frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}}\right)^\alpha$$

In order to estimate entry freshness, nodes maintain Δt_{since} and Δt_{alive} for each entry. Nodes learn and update these values in the following way:

- Nodes include their current Δt_{alive} in every packet they send. When a node directly communicates with its neighbor, it stores the received Δt_{alive} and sets Δt_{since} to 0.
- When a node learns of a new contact indirectly, it simply stores the received $(\Delta t_{alive}, \Delta t_{since})$ pair.
- If a node indirectly obtains a contact to a node already in the routing table, it examines the included $(\Delta t_{alive}, \Delta t_{since})$ pair. Smaller values of Δt_{since} indicate fresher information. Therefore, if the received Δt_{since} value is smaller than the one already stored in the routing table, it replaces the local values.

For the purpose of estimating entry freshness probabilities, nodes must determine the value of the Pareto distribution shape parameter, α . One strategy is to make a rough guess of the value of α and set it to a constant value of, for example, 1. A more elaborate way of dealing with an unknown value of the parameter is discussed in section 3.3.3 of [20].

2.4.5 Choosing eviction threshold

The eviction threshold $p_{threshold}$ directly affects the size of the routing table and, therefore, also the overall lookup time.

Setting $p_{threshold}$ too low results in many stale routing table entries, which in turn leads to unnecessary timeouts and inflated lookup times.

On the other hand, as the value of $p_{threshold}$ approaches the maximum value of 1.0, less entries survive the eviction process. Only the contacts that have been acquired recently and, as a result, have a very high estimated freshness probability remain in the table. These contacts are most certainly valid and timeouts during lookups are extremely rare. Nevertheless, nodes have few neighbors to forward search queries to. Consequently, the hop count increases invariably.

The authors of [19] examine the optimal values of $p_{threshold}$ for various network bandwidth budgets and values of β and α . They have come to the conclusion that the optimal value of $p_{threshold}$ for minimizing lookup time lies in $[0.8, 0.95]$ for all sensible combinations of bandwidth budget and network churn. Therefore, setting $p_{threshold}$ to 0.9 achieves near optimal lookup times.

2.4.6 Improved lookup

Accordion uses recursive routing (Figure 2.5) as opposed to iterative routing to speed up lookups and learn new routing entries. To further reduce lookup latency especially due to timeouts, nodes issue multiple copies of a lookup query in parallel. One copy is marked as primary and nodes always forward such lookups even if their budgets do not allow it.

Nodes individually control the degree of parallelism to meet their bandwidth budgets. Each node dynamically calculates the number of copies of lookups it may forward (w_p) based on the recent lookup rate. This parallelism window, w_p , is maintained in an additive increase and multiplicative decrease fashion.

ADJUST-PARALLELISM

```

1  if # of unique lookups forwarded < # of exploration packets sent
2      // Increment the parallelism window up to a maximum of 6
3       $w_p = \text{MIN}(w_p + 1, 6)$ 
4  elseif no exploration packets sent
5      // Quickly reduce overestimated values of  $w_p$ 
6       $w_p = w_p / 2$ 
```

On the one hand, if a node recently sent out more exploration packets than it forwarded unique lookups, the lookup activity is rather low and the parallelism window is increased by 1 up to a certain maximum value (e.g. 6). On the other hand, if there were no exploration packets sent out recently (because b_{avail} is negative), the node would exceed the bandwidth budget and must promptly reduce the level of parallelism by decreasing w_p by half.

GET-PARALLELISM

```

1  // Detected sudden surge of lookups
2  if  $b_{avail} \leq -b_{burst}$ 
3      // Temporarily disable parallel lookups
4      return 1
5  else
6      return  $w_p$ 
```

If there is a sudden surge in lookup traffic, a node must promptly reassess its lookup parallelism. Whenever a node suddenly depletes its bandwidth budget and b_{avail} reaches its minimum value of $-b_{burst}$, the node stops sending any exploration packets and temporarily disables sending parallel lookups; see GET-PARALLELISM. What is more, it drops any incoming lookup queries

which are not marked as primary and are, therefore, redundant.

Because lookups are issued in parallel along different paths, the probability that *all* next hop neighbors fail is much smaller than $1 - p_{threshold}$. Therefore, nodes may use a smaller neighbor eviction threshold p_{evict} than $p_{threshold}$ which depends on the current lookup parallelism w_p :

$$\begin{aligned} 1 - (1 - p_{evict})^{w_p} &= p_{threshold} \\ p_{evict} &= 1 - \sqrt[w_p]{1 - p_{threshold}} \\ p_{evict} &= 1 - e^{\frac{\ln(1-p_{threshold})}{w_p}} \end{aligned}$$

EVICT takes into account the current level of parallelism to delete routing entries.

EVICT(w_p)

- 1 // Determine the eviction threshold based on degree of parallelism w_p
- 2 $p_{evict} = 1 - e^{\frac{\ln(1-p_{threshold})}{w_p}}$
- 3 **for each** *entry* in *routing-table*
- 4 // Estimate the probability of the node being alive
- 5 $p = \text{entry}.\Delta t_{alive} / (\text{entry}.\Delta t_{alive} + \text{entry}.\Delta t_{since})$
- 6 **if** $p < p_{evict}$
- 7 delete *entry* from *routing-table*

When a node must route a lookup query it does not always forward it to the most closely preceding node of a search key in the routing table. Lookup queries are forwarded to neighbors with the highest expected density of contacts near the search key. Although this density is inversely proportional to the distance of the neighbor from the search key, it also depends on size of the neighbor's routing table. The larger the bandwidth budget of node, the larger the routing table. Therefore, lookups should be forwarded to neighbors with the largest value of v_i :

$$v_i = \frac{b_i}{d(n_i, k)} \quad (2.1)$$

where n_i is the i^{th} closest preceding neighbor of the lookup key k (i.e. n_1 is the immediate predecessor, n_2 precedes n_1 and so on) with a budget r_{avg} named b_i .

In addition, Accordion biases lookups toward physically closer neighbors and extends the progress metric (2.1) using network latency estimates:

$$v'_i = \frac{b_i}{d(n_i, k) \cdot \text{delay}(\text{this-node}, n_i)} \quad (2.2)$$

where $\text{delay}(\text{this-node}, n_i)$ is the estimated round-trip time from the local node to the neighbor (see chapter 4 on estimating network latencies).

To sum the lookup process up, when a node routes a lookup query, it considers a small fixed number of key predecessors (e.g. 6) and chooses w_p of them with the largest value of v'_i . Moreover, it considers only neighbors that bring the lookup at least half-way across the remaining distance to the search key if possible. The pseudo code of ACCORDION-LOOKUP and GET-NEXT-HOPS illustrates this process.

GET-NEXT-HOPS(*key*, *m*)

```

1  v = ∅
2  n = routing-table.GET-PREDECESSOR(key)
3
4  // Select the m best next hop from 8m closest predecessors of key
5  while |v| < 8 * m
6      distance = d(n.id, key)
7      // A hop must skip at least half the id distance to the key
8      if distance > d(this-node.id, key)/2
9          break
10     // Calculate how good a next hop node n would be using (2.2)
11     delay = ESTIMATE-RTT(this-node, n)
12     n.vi = n.ravg / (distance * delay)
13
14     v = v ∪ {n}
15     n = routing-table.GET-PREDECESSOR(n.id)
16 return first m nodes from v with the largest vi
```

ACCORDION-LOOKUP(q)

```

1  // Select 5 neighbors and piggy-back them in the acknowledgment
2  // so that the previous hop can learn new contacts.
3   $ack = \text{GET-NODES-TO-LEARN}(q.\text{prev\_hop}, q.\text{prev\_w}_p, q.\text{key}, 5)$ 
4  send an acknowledgment back to  $q.\text{prev\_hop}$  along with neighbors  $ack$ 
5
6  if this-node owns  $q.\text{key}$ 
7      return this-node IP address to query originator  $q.\text{initiator}$ 
8
9  // Determine the current degree of lookup parallelism
10  $parallelism = \text{GET-PARALLELISM}$ 
11
12 // Choose the best next hop neighbors using the progress metric (2.2)
13  $\text{routing-table.EVICT}(parallelism)$ 
14  $\text{next-hops} = \text{GET-NEXT-HOPS}(q.\text{key}, parallelism)$ 
15
16  $q.\text{prev\_hop} = \text{this-node}$ 
17  $q.\text{prev\_w}_p = parallelism$ 
18 // Forward the query in parallel
19 parallel for each  $n$  in  $\text{next-hops}$ 
20      $v = n.\text{ACCORDION-LOOKUP}(q)$ 
21     // Learn from acknowledgments
22      $\text{routing-table} = \text{routing-table} \cup v$ 

```

2.4.7 Assessment

Accordion's advantages:

- Scalability.
- It adapts to various network sizes and lookup patterns.
- Lookup latencies are on par with Chord (section 2.3) in high churn networks. In networks with negligible membership changes, Accordion provides lookup times comparable to One-hop (section 2.1).
- Network bandwidth control and locality awareness.

Disadvantages might include:

- Implementation complexity.

All in all, Accordion’s adaptability makes it suitable for overlay networks of all sizes. What is more, it gives users control over its network bandwidth usage, which is especially important for home users with limited upstream bandwidth.

2.5 Summary

This chapter gave an overview of some of the interesting ways of designing the key location layer. Although there are many other designs, Accordion is well suited for our needs. Accordion is both scalable and provides fast lookups in smaller DHTs with a stable set of participants. Therefore, it seems to be a good choice for a wide spectrum of DHT sizes.

Table 2.1: Overview of key location algorithms

Name	Hop count	Soft state	Notes
One-hop [13]	1	$\Theta(n)$	A complete routing table.
Tulip [5]	2	$O(\sqrt{n}\log(n))$	
Kelips [14]	$O(1)$	$\Theta(\sqrt{n})$	Loosely structured.
CAN [24]	$O(dn^{1/d})$	$\Theta(dr)$	Routes in a d -dimensional coordinate space.
Tapestry [33]	$O(\log(n))$	$\Theta(\log(n))$	
Pastry [27]	$O(\log(n))$	$\Theta(\log(n))$	Id prefix based lookup.
Bamboo [26]	$O(\log(n))$	$\Theta(\log(n))$	Based on Pastry.
Chord [30]	$O(\log(n))$	$\Theta(\log(n))$	
Kademlia [21]	$O(\log(n))$	$\Theta(\log(n))$	Parallel iterative lookups. Learns from lookups.
Koorde [16]	$O(\log(n))$	2	Based on de Bruijn graphs.
Accordion [20]	$O(\log(n))$	n/a	Adjusts the size of the routing tables for optimal performance.

Chapter 3

Data management

The data maintenance layer is responsible for storing and retrieving data associated with keys. It uses the lookup layer to find the node responsible for a key and it then directly communicates with the node to put or get the data.

The following two sections cover how the layer copes with node membership changes and mutable data .

3.1 Data availability

One of the main responsibilities of the data maintenance layer is to ensure data availability, i.e., the ability of the system to fetch data with high probability even as nodes join or leave the system. To achieve high availability, DHTs typically replicate data associated with a key to some of the neighbors of the node owning the key – the replica set of nodes. This set of nodes may be provided by the lookup layer, which can detect membership changes of the set and notify the data layer as nodes come and go. The key challenge is to keep track of the number of available replicas as the replica set changes.

Bamboo storage The data management system described in [25] was designed for the Bamboo key lookup algorithm [26]. The system disseminates key’s replicas to the neighbors of the key’s owner using an epidemic pull algorithm. Instead of actively tracking the number of replicas, nodes in a replica set periodically synchronize their key databases with each other and exchange replicas they are currently missing. The network overhead of key database synchronization is reduced by synchronizing entire intervals of keys by employing Merkle trees.

DHash replication The basic data replication scheme [7] developed for

Chord (section 2.3) is virtually the same as the one developed for Bamboo. To further increase availability, the system utilized erasure coding of large data blocks.

Total Recall The aim of Total Recall [6] is to automate the task of availability management. Users specify the desired level of availability and the system determines the appropriate number of replicas. The number of replicas is based on observations of availability of individual hosts of a replica set. Each key owner actively monitors the nodes of a replica set and allows a fraction of the nodes to be temporarily disconnected. New replicas are created only if the number of reachable replicas drops below a certain threshold. Allowing nodes to reintegrate their replicas into the system after a transient failure significantly reduces bandwidth consumption by the system.

Carbonite The main goal of Carbonite [8] is data durability, i.e., the ability of the system to eventually retrieve stored data even though the data may not be available right away. Similarly to Total Recall, Carbonite takes into account transient failures and allows nodes to reintegrate data back into the system. Unlike Total Recall, Carbonite does not attempt to predict node availability. The system preserves at least r_L replicas. A new replica is created if the current number of readable replicas falls below r_L . Up to a maximum of r_H replicas are tracked. As the analysis in [10] shows, the system ultimately creates such a number of replicas that most likely no new replicas will be created due to transient node failures.

In comparison with Total Recall, the user must specify the value of r_L instead of the desired overall availability. On the other hand, Carbonite arrives at the proper actual number of replicas naturally without the need to estimate host availability. All in all, Carbonite may consume a bit less bandwidth than Total Recall while providing the same level of data durability.

We have chosen to implement the basic replication scheme used in DHash without erasure coding. Firstly, DHash replication scheme is less complex than Carbonite’s complicated replica set synchronization and Total Recall’s host monitoring. Secondly, the design goal of the selected system coincides with our goal of high availability whereas Carbonite and Total Recall are targeted at applications durably storing large amounts of immutable data per node.

The rest of this section describes the DHash replica maintenance algorithm [7].

3.1.1 Replica maintenance

The replication system stores replicas on the first r nodes of a replica set which is a set of nodes provided and updated by the key lookup service. Each node maintains its replica set; therefore, all keys for which a node is responsible use the same replica set. The first node in a key's replica set is the owner of the key.

As the replica set changes, the system runs a global and a local maintenance protocol to restore the replica set to its ideal state. The local protocol recreates replicas within the replica set to ensure r copies of keys are present in the set. The global protocol moves displaced keys the node is no longer responsible for back to a node of the set.

Global maintenance protocol The global maintenance protocol periodically scans the entire key database for keys the current node should not store because it no longer belongs to the key's replica set. Such keys are first offered to nodes of the set. Next, any keys requested by the remote nodes are transferred from the current node. Finally, any remaining keys which the nodes of the replica set did not request and the current node is not responsible are discarded. To improve efficiency, keys are processed in entire intervals. GLOBAL-MAINTENANCE demonstrates the whole process in detail.

GLOBAL-MAINTENANCE

```

1  for ever
2       $key = key-db.GET-NEXT-KEY(key)$ 
3      // Let the lookup layer find the replica set of the  $key$ 
4      // and the interval of keys the set is responsible for
5       $(replica-set, key-interval) = LOOKUP-REPLICA-SET(key)$ 
6
7      // Determine if the key is displaced
8      if  $this-node.id \in replica-set[1..r]$ 
9           $key = key-interval.last$ 
10     else
11         for each  $node \in replica-set[1..r]$ 
12              $desired-keys = OFFER-KEYS(node, key-db[key-interval])$ 
13              $SEND-KEYS(node, key-db[desired-keys])$ 
14              $key-db.DELETE(desired-keys)$ 
15         // Delete remaining keys all nodes of the set already had
16          $key-db.DELETE(key-interval)$ 
17          $key = key-interval.last$ 
```

Local maintenance protocol Each node runs the local maintenance protocol in order to synchronize its key database with each node of its replica set. Both parties of database synchronization discover which keys they are missing. Missing keys are pulled from the other node, thereby recreating missing replicas.

LOCAL-MAINTENANCE

```

1  for ever
2      // Determine the replica set of the current node and
3      // the interval of keys it owns
4      (replica-set, key-interval) = GET-REPLICA-SET
5
6      for each node ∈ replica-set[2..r]
7          // Synchronize the key database with the remote node and
8          // pull missing keys in the background
9          SYNCHRONIZE(node, key-interval)

```

3.1.2 Key database synchronization

Hosts synchronize their databases of keys in order to identify which keys the hosts are missing. Unfortunately, exchanging lists of all the keys in the databases of both hosts would be very inefficient. Firstly, hosts may contain a large number of keys, so periodic synchronization would consume a great deal of bandwidth. Secondly, only a few keys may have been inserted since the last synchronization and, therefore, the bandwidth would be largely wasted on sending the same information.

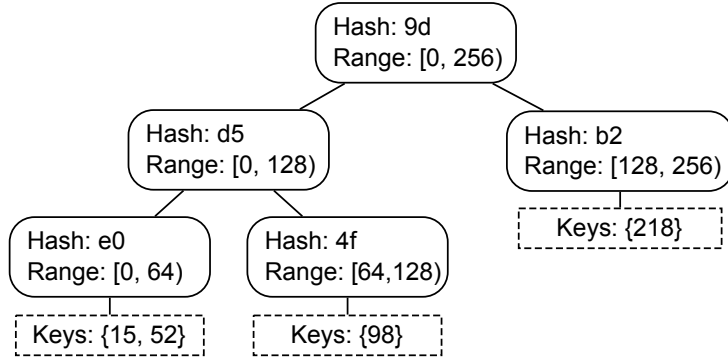
DHash replication system is optimized for the situation where both hosts store similar key databases. Instead of exchanging an unstructured list of keys, hosts synchronize a Merkle tree [22] of their keys (see Figure 3.1). A Merkle tree is a recursive tree of hashes. Internal nodes of the tree store a SHA-1 hash [15] of their child node hashes:

$$internal-node.hash = \text{SHA-1}(child[0].hash \circ \dots \circ child[31].hash)$$

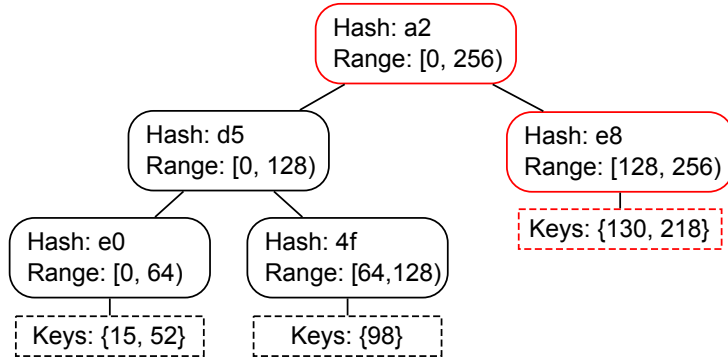
Similarly, a leaf stores the hash of all the keys the leaf refers to:

$$leaf.hash = \text{SHA-1}(key[0] \circ key[1] \circ \dots \circ key[31])$$

Each internal node contains the digest of a specific interval of the key space and if the tree uses a branching factor of 32, each of the node's 32 children summarizes one of the 32 equal-sized subranges of its parent's key range. For example, the root of the tree summarizes the entire key space $[0, 2^{160} - 1]$



(a) A simplified Merkle tree.



(b) The Merkle tree after 130 has been inserted.

Figure 3.1: The depicted Merkle trees have a branching factor of 2 and cover a reduced key space of $[0, 2^8 - 1]$. Each node contains the hash value of its child nodes as well as the key range the node summarizes with its hash value. The keys leaf nodes refer to are enclosed in a dashed rectangle. (a) The root node holds the hash value $9d$, which is the hash of $d5$ and $b2$. Similarly, the right child's hash value $b2$ is the result of hashing all of the keys the leaf refers to – in this case, it is a single key 218 . (b) This tree originated from tree (a) by inserting the key 130 . Note that hashes in the left subtree remained intact. Only the hash values of nodes along the path from the affected leaf to the root had to be updated (emphasized with red).

and the i^{th} child summarizes the subrange $[i \cdot 2^{160}/32, (i+1) \cdot 2^{160}/32 - 1]$. Generally, the leaves of the subtree rooted at the i^{th} tree node from the left at depth k refer only to keys from the interval $[i \cdot 2^{160}/32^k, (i+1) \cdot 2^{160}/32^k - 1]$.

This tree structure allows two hosts to compare entire ranges of the key space by exchanging the root nodes of the appropriate subtrees of the Merkle tree. If the root nodes contain equal hashes, both hosts store the same keys from the specific key range. If the hashes do not equal, hosts may recursively descend down the subtree and exchange child nodes to determine precisely which keys are missing.

What is more, inserting a key into the key database only changes the hashes along a single path from the key's leaf to the tree's root. Therefore, the rest of the tree remains largely intact.

SYNCHRONIZE(*host*, *range*)

1 SYNCHRONIZE-SUBTREE(*host*, *range*, *merkle-tree-root*)

SYNCHRONIZE-SUBTREE(*host*, *range*, n_{loc})

```

1   $n_{rem} = host.EXCHANGE-NODES(n_{loc}, range)$ 
2
3  // If we reached a leaf of any of trees, determine which keys
4  // we are missing. Otherwise, recurse down the tree.
5  if IS-LEAF( $n_{loc}$ ) or IS-LEAF( $n_{rem}$ )
6      FIND-MISSING-KEYS( $n_{loc}, n_{rem}, host, range$ )
7  else
8      for each pair ( $child_{loc}, child_{rem}$ )  $\in (n_{loc}.children, n_{rem}.children)$ 
9          // Recurse down the tree only if the subtrees differ.
10         if  $range \cap child_{loc}.range \neq \emptyset$  and  $child_{loc}.hash \neq child_{rem}.hash$ 
11             SYNCHRONIZE-SUBTREE(host, range,  $child_{loc}$ )
```

EXCHANGE-NODES($n_{rem}, range$)

```

1   $n_{loc} = merkle-tree.FIND(n_{rem})$ 
2  if IS-LEAF( $n_{loc}$ ) or IS-LEAF( $n_{rem}$ )
3      FIND-MISSING-KEYS( $n_{loc}, n_{rem}, remote-host, range$ )
4  return  $n_{loc}$ 
```

```

FIND-MISSING-KEYS( $n_{loc}, n_{rem}, host, range$ )
1  // If the remote tree node is a leaf, determine missing keys locally.
2  if IS-LEAF( $n_{rem}$ )
3      for each  $key \in n_{rem}.keys$ 
4          if  $key \in range$  and not  $key-db.INCLUDES(key)$ 
5              DOWNLOAD-MISSING-KEY( $key, host$ )
6  else
7      //  $n_{loc}$  is a leaf, but  $n_{rem}$  is a root of an entire subtree.
8      // As  $n_{loc}$  stores at most 32 keys, we are missing most of keys
9      // in the remote subtree – get a complete list of the keys in
10     // the range we are examining.
11      $remote-keys = host.GET-KEYS(range \cap n_{loc}.range)$ 
12      $missing-keys = remote-keys \setminus n_{loc}.keys$ 
13     for each  $key \in missing-keys$ 
14         DOWNLOAD-MISSING-KEY( $key, host$ )

```

Host h_a initiates synchronization with host h_b by calling SYNCHRONIZE. The host that initiated the synchronization controls the traversal of both the local and the remote host's Merkle tree. h_a starts off by exchanging the root node of its tree along with the hashes of the node's children with h_b in EXCHANGE-NODES. Next, h_a compares the hashes of the child nodes that lie in the key range that is being synchronized. If the hashes of the local and remote child nodes are equal, synchronization ends because both hosts store the same keys in the synchronized key range. Otherwise, h_a recursively descends down the subtree whose local hash differs from the remote node's hash as is shown on line 10 of SYNCHRONIZE-SUBTREE.

Recursion ends in FIND-MISSING-KEYS when h_a reaches a leaf in either of the trees. If the remote tree's leaf is reached first, it may be locally examined to determine which keys the current host is missing. If, on the other hand, the remote tree contains a complete subtree where the local tree ends with a leaf, all the keys in the examined range are downloaded from the remote subtree. The local leaf contains only a few keys and, therefore, the local host is missing most if not all the keys in the remote subtree.

Although host h_a traverses the trees, it exchanges the current node and, therefore, also the current position in the local Merkle tree with h_b . As a result, host h_b may also check which keys it is missing when the recursion reaches a leaf node in either of the trees (see EXCHANGE-NODES).

Even though hosts synchronize a specific interval of the key space, synchronization always proceeds from the root of the Merkle tree. Because of the large branching factor (e.g. 32) used in the tree, synchronization quickly descends down the correct subtree that contains the key range being syn-

chronized. To further speed up the process, hosts exchange not only a hash of a single tree node but also the hashes of all of its children.

3.2 Mutable data

The key database synchronization protocol described in the previous section does not consider the possibility of updating data. In order to introduce mutable data into the DHT, one must resolve the problem of deciding which of two given values is a more recent version of the stored data.

3.2.1 Bamboo updates

Even though the system presented in [26] does not support transparent updates of stored values, it allows stored values to be removed from the DHT with some cooperation from users.

When inserting a value into the table, users provide a deletion id, which is stored along with the original value. What is more, the system is extended to store values with unique key, deletion id pairs. Thus, there may be multiple values stored under the same key if they contain a different deletion id.

Moreover, the system is further extended to support a special type of values that denote a remove operation. When the system encounters a remove and an ordinary value associated with the same key and the same deletion id, the ordinary value is discarded in favor of the remove. Removing a value from the system is then a matter of storing a remove under the value's key and deletion id.

A remove must remain in the table even after all values stored in a key's replica set have been deleted. If the remove did not remain in the table, displaced replicas of the deleted values would be reinserted into the replica set of a key by the global maintenance protocol rendering the remove ineffective.

In order to avoid storing removes indefinitely, each put contains an associated time-to-live. Once the time-to-live expires, the value is deleted. Therefore, to successfully eradicate a value, users must put a remove with time-to-live set to a larger value than that of the value to be removed. Associating time-to-lives with puts has the additional advantage of eventually eliminating orphaned values that were put into the system by misbehaving nodes.

All in all, users may emulate an update of a value by first putting a remove with the present value's deletion id. Then, the updated value is inserted with a different deletion id. Unfortunately, until the remove absorbs the old value,

fetching values stored under the given key will return both the old and the new updated value.

Chapter 4

Estimating latencies

This chapter describes a system for estimating physical network latencies between nodes without having to directly measure the latency beforehand. Such a system is useful for both the data management layer as well as for the key location layer. In addition, it may be used to set proper retransmission timers.

For example, when a node is about to forward a lookup query, it may consider a set of next hops and forward the query to the node that is physically closest. The node may not have yet communicated with all of the next hop candidates and, therefore, may not have measured the round trip time to all the candidates. Unfortunately, measuring the network latencies and then sending the query to the closest node may take longer than forwarding the query to any next hop candidate right away. Similarly, a node fetching data should retrieve the replica that is located nearby, but the node has most likely never communicated with any of the nodes holding the replicas. In both of these situations the system discussed in this chapter may be used to locally estimate latencies to nodes the current node never contacted before.

Systems of estimating latencies are based on assigning coordinates to nodes. Coordinates are related to the actual latencies between nodes. The latency estimate is then derived from the distance between these virtual coordinates:

GNP Global Network Positioning [23] denotes a small set of N nodes as landmarks. Landmarks measure latencies between each other. The measured latencies are used to assign landmarks points in a $N - 1$ dimensional Euclidean space so that the distances between the points approximates the measured latencies with the smallest squared error. The landmarks are used as a frame of reference and other nodes calculate their coordinates in the Euclidean space from round trip time

measurements to the landmarks. Latency between two nodes is estimated as the Euclidean distance between the nodes' coordinates.

Virtual landmarks Authors of [31] decided to use physical distances from a larger random set of nodes as coordinates of a Euclidean space. Nodes physically near each other are likely to experience similar round trip times to the set of landmark nodes and, therefore, will be assigned similar Euclidean coordinates. The number of dimensions is significantly reduced by carefully transforming these coordinates into virtual coordinates without radically affecting accuracy. Estimating latency is simply a matter of calculating the Euclidean distance between two vectors holding the virtual coordinates.

In comparison to GNP, Virtual landmarks are less computationally demanding, provide similar or better latency estimates and the method scales to large networks.

Vivaldi The system proposed in [9] uses low-dimensional virtual coordinates and no landmark nodes. Nodes adjust their coordinates in order to roughly simulate a physical spring system. Rest spring lengths are set to known inter-node latencies and the actual spring lengths are derived from current node coordinates. Distances between virtual points are interpreted as the expected latencies between nodes. Decreasing the energy of the spring system through simulation reduces the squared error of latency estimates.

We have chosen Vivaldi to estimate inter-node latencies because it is fully decentralized, simple and computationally undemanding; yet it provides reasonably accurate estimates that are comparable to GNP [11]. In contrast to the other methods, Vivaldi integrates well with the key location service. As a result, latency predictions become a by-product of ordinary key lookups without noticeably increasing the network bandwidth usage.

4.1 The Vivaldi algorithm

Vivaldi [9] assigns nodes synthetic coordinates, so that the distance between two nodes' coordinates can be used to predict actual network latency between the nodes. Each node adjusts its coordinates as new round-trip time (RTT) measurements to other nodes become available.

Because coordinates lie in a metric space and inter-host latencies do not adhere to the triangle inequality, Vivaldi cannot predict latencies exactly.

Therefore, it aims to minimize the squared error of predictions:

$$E = \sum_i \sum_j (L_{ij} - \|\vec{x}_i - \vec{x}_j\|^2)$$

where L_{ij} is the measured RTT between nodes with coordinates \vec{x}_i and \vec{x}_j .

Minimizing the squared error is analogous to minimizing the potential energy of a mass-spring system, which is proportional to the square of spring displacement. Therefore, Vivaldi roughly simulates a spring system and adjusts node coordinates as if springs were placed between nodes. Rest spring lengths are set to measured RTTs and the current spring lengths are considered to be the current distances between node coordinates. To simplify and speed up the simulation, Vivaldi ignores mass or momentum of nodes and springs and instead moves a node in the coordinate space in the direction of a force that would have been exerted by the springs in small steps at a time.

4.1.1 Updating estimated error

Each node maintains an estimate of the relative error (e) of the latency predictions that use the node's current coordinates (\vec{x}). It is based on recent RTT measurements and it is maintained as an exponentially weighted moving average of the relative error of the predicted and measured RTT.

Highly inaccurate predictions of latencies from the local node to many other nodes suggest an incorrect placement of the current node in the coordinate space as opposed to the placement of the sampled nodes. Such erroneous predictions lead to a large estimated relative error. Therefore, the estimated relative error characterizes how certain the node is about its coordinates.

UPDATE-ESTIMATED-ERROR captures how the current node updates its estimated error after a new RTT measurement of r_{tt} to a node with coordinates \vec{x}_{rem} and relative error e_{rem} becomes available.

VIVALDI-UPDATE($r_{tt}, \vec{x}_{rem}, e_{rem}$)

- 1 UPDATE-COORDINATES($r_{tt}, \vec{x}_{rem}, e_{rem}$)
- 2 UPDATE-ESTIMATED-ERROR($r_{tt}, \vec{x}_{rem}, e_{rem}$)

UPDATE-ESTIMATED-ERROR($r_{tt}, \vec{x}_{rem}, e_{rem}$)

- 1 // Sample weight w divides the error between the local and remote node
- 2 $w = e_{loc} / (e_{loc} + e_{rem})$
- 3 // Compute the relative error of this measurement
- 4 $e_{sample} = \left| \|\vec{x}_{rem} - \vec{x}_{loc}\| - r_{tt} \right| / r_{tt}$
- 5 // Update the local estimated relative error (4.1)
- 6 $e_{loc} = c_e \cdot w \cdot e_{sample} + (1 - c_e) \cdot (1 - w) \cdot e_{loc}$

The estimated relative error is maintained as an exponentially weighted moving average to smooth the estimate of the error:

$$e'_{loc} = e_{sample} \cdot \frac{e_{loc}}{e_{loc} + e_{rem}} + e_{loc} \cdot \frac{e_{rem}}{e_{loc} + e_{rem}} \quad (4.1)$$

The relative error from the most recent RTT measurement is e_{sample} . e_{loc} and e_{rem} are the local and remote node's error. If the local error is small in comparison with the sampled node's error, the local error changes only slightly as the prediction error (e_{sample}) is most likely due to suboptimal coordinates of the remote node. On the other hand, if e_{loc} is large as opposed to e_{rem} , the coefficient next to e_{sample} will be large and the sample error e_{sample} will be accounted.

Unlike in (4.1), UPDATE-ESTIMATED-ERROR uses an additional constant tuning factor c_e .

4.1.2 Updating coordinates

UPDATE-COORDINATES depicts how the algorithm updates local node's coordinates \vec{x}_{loc} when a round-trip time of rtt is measured to a node with coordinates \vec{x}_{rem} and estimated relative error e_{rem} .

UPDATE-COORDINATES($rtt, \vec{x}_{rem}, e_{rem}$)

- 1 // Sample weight w divides the error between the local and remote node
- 2 $w = e_{loc} / (e_{loc} + e_{rem})$
- 3 // Compute the step size
- 4 $\delta = c_e \cdot w$
- 5 // Update local coordinates ($u(\vec{y})$ is a unit vector of \vec{y})
- 6 $\vec{x}_{loc} = \vec{x}_{loc} + \delta \cdot (\|\vec{x}_{rem} - \vec{x}_{loc}\| - rtt) \cdot u(\vec{x}_{rem} - \vec{x}_{loc})$

On line 2, the sample weight w is calculated as:

$$w = \frac{e_{loc}}{e_{loc} + e_{rem}}$$

Sample weight is used to divide the error between the local and the remote node. If the relative error of the local coordinates e_{loc} is large compared to remote node's relative error e_{rem} , the local node's coordinates are most likely to blame for the difference between the predicted and actual RTT and should be adjusted. On the other hand, if e_{loc} is small compared to e_{rem} , the local coordinates should be adjusted only slightly because incorrect remote node's coordinates are most likely the cause of the wrong RTT prediction.

Therefore, this node is moved in the coordinate space by a step δ (line 4) that is proportional to w :

$$\delta = c_c \cdot \frac{e_{loc}}{e_{loc} + e_{rem}}$$

where c_c is a constant conversion factor.

On line 6 the local node is moved by a fraction δ of the force exerted by a spring placed between the local node \vec{x}_{loc} and the remote node \vec{x}_{rem} :

$$\vec{x}'_{loc} = \vec{x}_{loc} + \delta \cdot (\|\vec{x}_{rem} - \vec{x}_{loc}\| - rtt) \cdot u(\vec{x}_{rem} - \vec{x}_{loc})$$

where $u(\vec{x}_{rem} - \vec{x}_{loc})$ denotes a unit vector pointing in the direction of the remote node and $(\|\vec{x}_{rem} - \vec{x}_{loc}\| - rtt)$ is the displacement from the rest length of the spring.

Moving the node by step δ has many desired properties. When a node starts up Vivaldi, its estimated error is large and it, therefore, quickly moves by large steps to a better position in the coordinates space. Moreover, the new node does not disrupt positions of older nodes which have already arrived at coordinates that predict latencies well. Older nodes with good coordinates will have a small estimated error compared to the new node. As a result, the older nodes will only move by minuscule steps.

Chapter 5

Implemented DHT

We have implemented Kontajner which is a DHT that utilizes the Accordion lookup algorithm as described in section 2.4 and the DHash replication scheme presented in section 3.1.

The resulting C++ implementation makes use of Boost [1], which is a large and mature collection of well documented C++ libraries. In particular, we have used the following libraries:

Boost.Asio Asio is an input/output library that allows users to easily create asynchronous networking applications. We have chosen Asio to form the basis of our DHT implementation because its asynchronous mode of operation matches the asynchronous nature of DHT operation well.

Boost.Serialization User values put into Kontajner are serialized by the Serialization library. The library supports various platforms and compilers and offers an easy way to add serialization to complete C++ classes. What is more, it is capable of serializing STL¹ containers out of the box and it handles the serialization of polymorphic types through pointers to base classes with little additional effort.

log4cxx Every node in a DHT may be simultaneously involved in multiple key lookup or key synchronization operations. As the number of nodes in the overlay increases, the system becomes increasingly difficult to debug. Stepping through code may not easily reveal the true source of a bug, which may be located many overlay hops away from the point where the error first manifests itself. Moreover, stopping at a breakpoint may inadvertently change the timing of DHT operations or cause retransmission timers to time out, which further complicates debugging. Therefore, we have decided to add logging to core components of

¹C++ Standard Template Library

Kontajner via the means of the `log4cxx` [3] library. `log4cxx` is a port of the popular Java logging library `log4j`.

5.1 User interface

Kontajner is exposed to the user via the *kontajner* class. To satisfy a broader range of use cases, *kontajner* provides to the users three different ways of interfacing with the underlying DHT.

Table 5.1: Kontajner conceptual API

Operation	Description
create	Creates a new Kontajner DHT where the current node is the only participant of the table.
join	Joins an existing Kontajner DHT.
leave	Correctly leaves an existing Kontajner DHT notifying other participants of the overlay.
abort	Immediately leaves the overlay without notifying any members of the overlay of the event.
put	Stores a new value in the table or it extends the time to live of an already present value.
remove	Stores a remove in the table. Any values matching the remove will be eventually eradicated from the table.
get	Retrieves all values stored under the given key.

Synchronous interface The synchronous interface represents the most intuitive way of accessing the DHT. Every operation blocks until it completes, so this interface may be used in situations where small time lags are not an issue. The following illustrates how one might create a new Kontajner DHT and store a string under the key 7.

```
// Local IP address and port to communicate through.
node_endpoint endpoint(ip_address , port);
kontajner dht(endpoint);

// Establish a new Kontajner DHT
dht.create();

// Prepare data we wish to insert.
key_id key(7);
```

```

std::string str("All Your Base");

// Store the string under the key with time-to-live
// set to 60 seconds and a random deletion token.
dht.put(key, str, seconds(60), id_t::random_id());

// Lets retrieve the stored string
std::string str2;
dht.get(key, str2);

```

Futures interface The blocking behavior of the synchronous interface and the associated time lags necessary for completion of operations may not be acceptable in some applications. Applications requiring nonblocking access may utilize the futures interface. All operations exit immediately and return a future object. The future object may be queried at any time to determine whether the associated operation has completed. In any case, if the user attempts to retrieve the result of an operation from the future object, the call blocks until the result is available. The following listing demonstrates how to initiate the retrieval of values stored under two different keys and later wait for both operations to complete.

```

// Initiate fetch of values under key 1 and 2.
get_future fut1 = dht.future_get(key_id(1));
get_future fut2 = dht.future_get(key_id(2));

std::cout << "Get in progress ...";

int num1, num2;
// Now wait for both fetches to complete.
fut1.get(num1);
fut2.get(num2);

```

Asynchronous interface The main drawback of the futures interface is the need to poll regularly whether an operation has completed if the user does not wish to block under any circumstances. In this use case, one should employ the asynchronous interface. It allows users to supply a callback function that will be invoked once an operation completes. The completion handler executes in a separate thread, so users must use proper synchronization primitives when transferring the result back to the user thread.

```

void func(const throwable_error & error)
{

```



```

        if(error){
            std::cout << "Leaft with error " << error;
        }
    }

    // Leave the DHT and when done, call func.
    dht.async_leave(func);

```

Unlike ordinary STL containers, values of varying types may be stored in *kontajner*. Users are discouraged from creating separate DHTs for separate value types because it would waste resources—most notably—network bandwidth.

5.2 Design

Kontajner was designed with flexibility in mind and it is not tied to a single combination of lookup and storage algorithms. The subsystems that Kontajner consists of communicate with each other via well defined interfaces:

async::service This service allows other classes to issue asynchronous function calls. What is more, the service can create timers that invoke their time out handlers in an asynchronous fashion.

msg::transport Classes implementing the message transport interface provide upper layers of the DHT with the ability to transport messages between nodes. Because the message transport is shared between other subsystems, its main responsibility is to demultiplex received messages and dispatch them to the correct destination subsystem.

async_lookup Lookup algorithms are represented as classes implementing the *async_lookup* interface. This interface enables other layers to determine the range of keys owned by the local node and to find the replica set of a key. Lookup algorithms typically use the services of message transport to communicate between nodes.

async_timed_storage Data maintenance algorithms are required to store unstructured memory blocks in a reliable way and to expose this functionality via the *async_timed_storage* interface.

This design, for example, allows to change the lookup algorithm used by Kontajner by simply instantiating a different lookup class. What is more, custom message transport and asynchronous service classes may form the basis of a simulator of Kontajner's operation.

5.3 Implementation

Instead of creating a new thread for each new request, the subsystems implement their operations with asynchronous function calls. Invoked functions exit immediately and signal completion by calling a callback function. As a result, multiple operations may be executed concurrently in a single thread.

In comparison with a multi-threaded approach, this design is inherently thread safe. In addition, it fits well with the asynchronous character of communication in the overlay network.

On the other hand, chaining multiple callback functions results in less obvious code paths. Moreover, in order not to interfere with the user thread, functions must execute in a separate background thread. Consequently, a form of thread safe marshaling of requests from the user thread to the background thread is still required.

5.3.1 RPCs and message transport

The default message transport provides best effort message delivery and works on top of the UDP protocol. In contrast to TCP, UDP is able to send short messages immediately without having to first establish a connection. While this mode of operation suits the lookup layer well, it is not sufficient for the data maintenance layer.

Different subsystems require different delivery guarantees of requests to other nodes. Therefore, instead of creating a universal remote procedure call (RPC) system, each subsystem creates custom proxy and stub objects which transport RPCs using messages exactly the way required by the subsystem.

5.3.2 Lookup

The Accordion lookup algorithm is encapsulated by the *accordion_lookup* class. Unlike the algorithm presented in section 2.4, our implementation does not eagerly evict entries from the routing table. Contacts to nodes that are deemed dead based on the current level of lookup parallelism are only hidden by the routing table. As result, no routing entries are mistakenly deleted if the degree of parallelism temporarily drops or suddenly fluctuates.

5.3.3 Data maintenance

The *simple_bamboo* class encompasses the data maintenance layer.

Putting a value into the DHT consists of first looking up the key's replica set using the lookup layer. Next, *simple_bamboo* sends a copy of the value

to each node of the replica set. If any of these nodes accepts the value, the operation completes successfully.

The get operation also starts by looking up the replica set of the search key. The local node then contacts the physically closest node of the replica set as predicted by the latency estimation subsystem and downloads all the values associated with the key stored at that node.

5.3.4 Kontajner

Users access the implemented DHT with the *kontajner* class. The main responsibility of this class is to instantiate and glue together the individual subsystems. Moreover, it manages the background thread where subsystems execute their operations asynchronously. User requests are marshaled to the background thread by adding the request into the thread's queue of pending asynchronous calls.

Kontajner's asynchronous interface is the class's native interface. The futures interface is built in terms of the asynchronous interface and the synchronous interface is in turn implemented on top of the futures interface.

Chapter 6

Summary

6.1 Conclusion

The primary goal of this thesis was to give an overview of the various parts that distributed hash tables consist of. Secondly, the thesis also aimed at implementing a simple-to-use distributed hash table in C++ that would minimize lookup times and maximize resiliency all the while having a moderate overhead and being able to scale to potentially large networks.

Firstly, we briefly described the relationship between the two main layers of a DHT, namely the key lookup layer and the data maintenance layer. Secondly, we discussed multiple key lookup algorithms and the issues these algorithms must deal with. What is more, we highlighted the strengths and weaknesses of the individual algorithms. The Accordion lookup algorithm was selected as the basis for our distributed hash table implementation because of its ability to adapt to different work loads and still provide fast lookups.

Thirdly, we focused on the data maintenance layer and concisely summarized the available data replication techniques. DHash style replication was selected for our DHT implementation as its properties best aligned with our goals of high availability of stored data. The selected replication system was, then, studied in greater detail. Furthermore, we explored the alternatives of adding support for mutable data. Although Bamboo's support of mutable data does not provide a completely transparent way of updating the stored data, it is still very useful. Consequently, we adopted Bamboo's mutable data support in our DHT implementation.

In addition, we examined the problem of estimating latencies to nodes that have never been contacted before. The main focus was on the Vivaldi algorithm because unlike the other available systems, it is completely decen-

tralized.

Last but not least, we presented the resulting C++ distributed hash table implementation – Kontajner. Kontajner hides the entire DHT machinery behind a simple-to-use interface. To demonstrate the simplicity of the provided interface, we included small code snippets of Kontajner’s usage. Additionally, Kontajner’s internal design is flexible enough, so that the individual subsystems that compose Kontajner can be easily swapped for better future implementations.

Although there are many other algorithms available for structuring DHTs, we believe we succeeded in giving an overview of the broad spectrum of issues associated with DHT operation and of the techniques used to tackle these issues.

We are also convinced that the selected algorithms strike a good balance between reasonable performance and resiliency on the one hand, and moderate network usage on the other. However, the performance of the implementation remains to be evaluated in a real deployment on hundreds of nodes. A more practical approach to evaluating Kontajner’s performance would be to simulate Kontajner’s operation by replacing both the message transport and the asynchronous service with a simulation layer. In any case, evaluating the performance characteristics of Kontajner remains subject to future work.

6.2 Future work

There are many ways in which the current implementation can be improved and extended.

Firstly, more key location algorithms could be included to give knowledgeable users the option of choosing the location algorithm that best meets their specific needs.

Secondly, network traffic could be reduced by supporting erasure coding of large data blocks at the data replication layer [7]. Another approach to decreasing network traffic is to consider transient node failures and to allow nodes to reintegrate their data into a DHT after being temporarily disconnected.

A major extension to the current implementation would be to expose the key location layer via a key based routing interface [12]. Key based routing can form the basis of decentralized object location and routing, and group multicast services.

Finally, future versions could tackle security problems in distributed hash tables. Security of DHTs is still part of active research and current surveys

of the various techniques available [32] suggest that securing DHTs in environments with malicious users is very difficult.

Bibliography

- [1] Boost. <http://www.boost.org/>.
- [2] libtorrent. <http://www.rasterbar.com/products/libtorrent/>.
- [3] log4cxx. <http://logging.apache.org/log4cxx/index.html>.
- [4] The Circle. <http://savannah.nongnu.org/projects/circle/>.
- [5] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, and S. Ron. Practical locality-awareness for large scale information sharing, 2005.
- [6] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: system support for automated availability management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 25–25, Berkeley, CA, USA, 2004. USENIX Association.
- [7] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.
- [8] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.
- [9] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. *SIGCOMM Comput. Commun. Rev.*, 34(1):113–118, 2004.
- [10] Frank Dabek. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, November 2005.

- [11] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2004. ACM.
- [12] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiawicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.
- [13] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 2–2, Berkeley, CA, USA, 2003. USENIX Association.
- [14] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.
- [15] National institute of standards and technology. Fips 180-2, secure hash standard, federal information processing standard (fips), publication 180-2. Technical report, Department of Commerce, August 2002.
- [16] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. pages 98–107, 2003.
- [17] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.
- [18] Jon Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170, New York, NY, USA, 2000. ACM.
- [19] Jinyang Li. *Routing tradeoffs in dynamic peer-to-peer networks*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006. Adviser-Morris, Robert.

- [20] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of dht routing tables. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 99–114, Berkeley, CA, USA, 2005. USENIX Association.
- [21] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.
- [22] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378, London, UK, 1988. Springer-Verlag.
- [23] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *In INFOCOM*, volume 1, pages 170–179, 2002.
- [24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.
- [25] Sean Rhea. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, 2005.
- [26] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiawicz. Handling churn in a dht. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [27] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.
- [28] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multi-media Syst.*, 9(2):170–184, 2003.

- [29] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.
- [30] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.
- [31] Liying Tang and Mark Crovella. Virtual landmarks for the internet. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 143–152, New York, NY, USA, 2003. ACM.
- [32] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. A survey of DHT security techniques. *ACM Computing Surveys*, 2009. http://www.globule.org/publi/SDST_acmcs2009.html, to appear.
- [33] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA, 2001.